

# **ratecalc**

November 29, 2017  
17:13

## **Contents**

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Input for RATECALC</b>	<b>5</b>
<b>3</b>	<b>Fill Cross Section Tables</b>	<b>15</b>
<b>4</b>	<b>Integrands for Rate Calculations</b>	<b>22</b>
<b>5</b>	<b>ALADDIN Related Routines</b>	<b>27</b>
<b>6</b>	<b>Integration Routines</b>	<b>32</b>
<b>7</b>	<b>References</b>	<b>46</b>
<b>8</b>	<b>INDEX</b>	<b>48</b>

## 1 Introduction

\$Id: dcab6b0018d732c3a52610f4ba3f0953d134c02d \$

The plasma source terms  $S_{A_p,k}$  due to interactions between neutral atoms and charged particles[1], can be written in the form

$$S_{A_p,k} = \Gamma \int d^3v f(\vec{v}) \langle \sigma_k(v_{rel}) v_{rel} \Delta A_p \rangle_p, \quad (1)$$

where  $\Gamma$  is the flux of neutral particles,  $f(\vec{v})$  is the neutral particle distribution function,  $p$  is a plasma species index,  $\sigma_k$  is the cross section for the  $k$ th reaction between the neutral and species  $p$ ,  $v_{rel}$  is the relative velocity of the two particles, and  $\Delta A_p$  is the change in a plasma quantity (e.g., density, momentum, energy, etc.) in a single collision. The angle brackets denote an average over the plasma species' distribution function.

By expressing  $\Delta A_p$  as a power series in the relative velocity, the  $S_{A_p,k}$  can be written as a linear sum of velocity moments of the distribution function weighted by the quantity in angle brackets. The latter, in turn, is a (plasma species) velocity moment of the reaction rate associated with  $\sigma_k$ . The primary purpose of this code is to compute these moments at user-specified parameters using cross section data accessed from external files.

Presently, the cross section data is assumed to reside in the ALADDIN database. The formats of the input file and common blocks reflect this assumption. The input information used to identify the data in the database are in the *ra* class. Output from this code is stored in netCDF files using the *xs* class (see file classes.web for descriptions of both classes).

If the requested data represent a fit, the parameters used to describe the fit and the name of the function required to evaluate it are retrieved from the database and stored in the output data file. Alternatively, if the data for a particular dependent variable are in a “table”, the input information specifies the range, number, and spacing of the corresponding independent variables. At each set of independent variables, the dependent variable (cross section, reaction rate, or higher moment) is evaluated and sent to the output file.

More specifically, there are now three types of tabular organizations:

**table\_calc** Is of the type described above. Cross sections are averaged over a distribution function to obtain reaction rates and higher moments.

**table\_eval** Is used to simply evaluate the ALADDIN fit and store the data (for, e.g., transferring cross section data or ALADDIN reaction rates).

**table\_external** Is used to handle extraneous (usually constants) data which cannot be obtained from ALADDIN or which might be peculiar to a specific reaction. Placeholders are generated in the netCDF files; the user can later insert manually the required data.

The integral in Eq. (1) is evaluated here in two different regimes. The first is applicable to situations in which a neutral atom (“test”) collides with plasma electrons (“background”). In this case, the velocity moments required are given by

$$\langle \sigma v v^n \rangle = \frac{4\pi}{\alpha m} \left( \frac{2}{m} \right)^{(n+2)/2} \int_0^\infty dE \sigma f(E) E^{(n+2)/2}, \quad (2)$$

where  $\sigma(E)$  is the cross section,  $f(E)$  is an arbitrary distribution function,  $m$  is the particle mass relating  $E$  and  $v$  via  $E = \frac{1}{2}mv^2$ , and  $\alpha$  is the normalization constant for the distribution function,

$$\alpha = \frac{4\pi}{m} \left( \frac{2}{m} \right)^{1/2} \int_0^\infty dE f(E) E^{1/2}.$$

In order for the numerical integrations to work effectively, it is necessary that  $E$  be in practical units; eV in this case.

More specifically, the assumptions made in arriving at Eq. (2) are:

1.  $\sigma$  is a function only of the magnitude of the relative velocities of the colliding particles.
2. The distribution function is a function only of the particle’s energy (or, equivalently, speed).
3. The velocity of the “test” particle is much less than the typical velocity of the “background” particles (i.e., defined by  $f$ ).

In the second regime, the velocities of the two colliding particles may be comparable. For example, this would be the case if both species were “heavy” (e.g., neither was an electron). The integration in Eq. (1) must then be done over the relative velocity. This can be done conveniently when the plasma species is described by a Maxwellian distribution function. It is conceivable that an arbitrary distribution could be specified in terms of orthogonal polynomials times a Maxwellian so that the required integral could be computed as a linear sum of integrals like those described here.

The velocity moments in this regime are expressed in terms of the standard integral[1]

$$I_{\ell,n} \equiv \frac{\exp(-u^2/u_p^2)}{\sqrt{\pi} u u_p^{n+1}} \int_0^\infty dv_{rel} v_{rel}^{2+n} \sigma_\ell(v_{rel}) \left\{ \exp \left[ \frac{-v_{rel}(v_{rel} - 2u)}{u_p^2} \right] - (-1)^n \exp \left[ \frac{-v_{rel}(v_{rel} + 2u)}{u_p^2} \right] \right\}, \quad (3)$$

where  $u$  is the velocity of the neutral particle, and  $u_p$  is the thermal velocity of the plasma particle. The latter is related to the plasma temperature by  $T_p = \frac{1}{2}m_1 u_p^2$ ; the former is derived from the neutral energy in the same way. The subscript  $\ell$  is included to allow the treatment of elastic collisions ( $\ell \neq 0$ ). Note that Eq. (3) is not the same as  $\langle \sigma v v^n \rangle$  because of the need to treat angular dependence in the integrand of Eq. (1), e.g., for the momentum source rate. Quantities such as this can be found as a linear combination of the  $I_{\ell,n}$ . The results provided by this code is  $I_{\ell,n} u_p^n$ . Note that  $\ell = 0$  represents the “total” cross section for either inelastic or elastic collisions. In either case, its  $n = 0$  moment is simply referred to as the “reaction rate”, which is used in computing the local probability of the reaction. For elastic collisions,  $\ell = 1$  represents the diffusion cross section and is associated with momentum transport; its  $n = 0$  moment is designated “I\_1\_0” to distinguish it from the reaction rate. Higher moments are denoted by I\_0\_1\*up or I\_1\_1\*up ( $n = 0, \ell = 0$  or 1) and I\_0\_2\*up^2 or I\_1\_2\*up^2 ( $n = 1, \ell = 0$  or 1). There is also I\_2\_4\*up^4 ( $n = 4, \ell = 2$ ) that has thus far been used only outside of DEGAS 2 for the purpose of computing viscosity. Note that  $I_{\ell,n} u_p^n$  has the same dimensions as  $\langle \sigma v v^n \rangle$ . In practice, moments  $n > 0$  for  $\langle \sigma v v^n \rangle$  (i.e., for electron background) are not used.

In the expressions defining the neutral energy and plasma temperature, the mass  $m_1$  used is 1 amu. The independent variable is referred to as “specific energy” or “specific temperature” to indicate this. This allows the results to be easily rescaled for different isotopes. It is crucial to note the asymmetry between the neutral and plasma variables. The rates computed by this code are intended to be used primarily in Monte Carlo neutral transport codes. The objective of these codes is to, in some approximation, compute the neutral particle distribution function. Hence, this information is not known until the end of the calculation; in a Monte Carlo code, however, one will know the instantaneous *energy* of a test neutral particle. To carry out the kinetic calculation, the distribution of the background species must be known. This is usually assumed to be Maxwellian; the characteristic parameter in this case is the plasma *temperature*. So, it is usually the case that the independent variable “specific temperature” is to be associated with a plasma ion species, “specific energy” with a neutral species, and “temperature” with plasma electrons.

To facilitate numerical integration, the units used for the various velocities are  $\sqrt{\text{eV}/\text{amu}}$ ; neither  $u$  nor  $u_p$  should be  $\leq 0$ . However, the energy used as the argument for  $\sigma$  must have a specific mass. Its value (specified by *ra\_mass* here) varies, depending on the origin of the cross section data; it is assumed to be in grams. The independent variable to be used with cross sections in these reactions should be “specific\_energy”. This will eliminate any confusion in subsequent uses of the data. An independent variable of “energy” should be used for electron impact processes, or other reactions in which the mass to use is clear.

For example, for the data described in Ref. [2] one would need to take *ra\_mass* to be that of the charged reagent (electron or ion). On the other hand, the data in Ref. [3] use a mass of 1 amu for “heavy” particle reactions and the electron mass for electron impact processes.

```
"ratecalc.f" 1≡
@m FILE 'ratecalc.web'
```

The unnamed module. This subroutine just extracts the input filename from the argument line and passes it along to the reading subroutine.

```
"ratecalc.f" 1.1 ≡
  program ratecalc
    implicit none_f77
    implicit none_f90
    character*LINELEN arg
    character*FILELEN filename

    sy_decls

    assert(arg_count() ≡ 1)
    call command_arg(1, arg)
    filename = arg
    call read_input(filename)

    stop
  end
```

⟨ Functions and subroutines 2 ⟩

## 2 Input for RATECALC

Read input data into common variables and set up defaults. The format of the input file is described in the examples. Note that some quantities are either determined locally or assumed to have specific values. Those variables are:

*ra\_num*\* Set by counting the corresponding labels.

*xs\_num\_dep\_var* Set by counting the number of dependent variables specified.

*xs\_rank* Set by counting the number of independent variables.

- Note that when the data are a fit, the *xs\_table* array will effectively be of rank one, i.e., containing the fit coefficients.
- Also in this case *xs\_tab\_index* is set temporarily to the parameter used to specify the maximum number of fit coefficients since the user will not know this in general.

*xs\_units* Are assumed to be cgs or eV for temperature and energy.

*xs\_multiplier* Are then the corresponding MKS conversion factors.

```

⟨ Functions and subroutines 2 ⟩ ≡
subroutine read_input(filename)
  implicit none_f77
  ra_common
  xs_common
  implicit none_f90

  character*FILELEN filename // Input
  integer length // Local
  character*LINELEN line

  st_decls
  xs_decls

  open(unit = diskin, file = filename, form = 'formatted', status = 'old')
  if ( $\neg$ read_string(diskin, line, length)) then
    line = 'END'
  end if
  length = parse_string(line(: length))

loop1: continue
  if (line ≡ 'END')
    goto eof
  if (line ≡ 'DEPENDENT_VARIABLES') then
    call read_dependent(diskin, line)
  else if (line ≡ 'INDEPENDENT_VARIABLES') then
    call read_indep(diskin, line)
  else if (line ≡ 'REACTIONS') then
    call read_reaction_list(diskin, line)
  end if

  goto loop1

eof: continue
  return
end

```

See also sections 2.1, 2.2, 2.3, 3, 3.1, 3.2, 4, 4.1, 4.2, 4.3, 4.4, 5, 5.1, 6, 6.1, 6.2, 6.3, and 6.4.

This code is used in section 1.1.

Read in data for dependent variables.

```

⟨ Functions and subroutines 2 ⟩ +≡
  subroutine read_dependent(unit, line)
    implicit none f77
    xs_common
    implicit none f90
    integer unit      // Input
    character*(*) line      // Output
    integer length, p, b, e, i, j      // Local
    st_decls
    xs_decls

    xs_num_dep_var = 0

next_dep_var: continue // Label macros are defined in ratecalc.hweb
    assert(read_string(unit, line, length))
    assert(length ≤ len(line))
    length = parse_string(line(: length))
    if (line ≡ 'INDEPENDENT↓VARIABLES')
        goto end_dep_var

    p = 0 // For each dep. var.:
    assert(next_token(line(: length), b, e, p)) // Read name
    xs_num_dep_var ++
    assert(xs_num_dep_var ≤ xs_dep_var_max)
    xs_var0, xs_num_dep_var = read_text(line(b : e))

    assert(next_token(line(: length), b, e, p))
    xs_eval_name_xs_num_dep_var = read_text(line(b : e)) // Eval. name

    if (next_token(line(: length), b, e, p)) then
        xs_spacing0, xs_num_dep_var = read_text(line(b : e)) // And spacing
    else
        xs_spacing0, xs_num_dep_var = 'unknown'
    end if

    goto next_dep_var

end_dep_var: continue
    /* Utilize cgs assumptions for units. Multipliers are conversions to MKS. */
    do i = 1, xs_num_dep_var
        if (xs_var0, i ≡ 'cross_section') then
            xs_units0, i = 'cm²',
            xs_mult0, i = const(1., -4)
        else if (xs_var0, i ≡ 'reaction_rate') then // I0,0
            xs_units0, i = 'cm³/s'
            xs_mult0, i = const(1., -6)
        else if (xs_var0, i ≡ 'I_1_0' ∨ xs_var0, i ≡ 'I_1_0') then
            xs_units0, i = 'cm³/s'
            xs_mult0, i = const(1., -6)
        else if (xs_var0, i ≡ 'I_1_1*up' ∨ xs_var0, i ≡ 'I_0_1*up' ∨ xs_var0, i ≡ 'I_1_1*up') then
            xs_units0, i = 'cm⁴/s²',
            xs_mult0, i = const(1., -8)
        else if (xs_var0, i ≡ 'I_1_2*up²' ∨ xs_var0, i ≡ 'I_0_2*up²' ∨ xs_var0, i ≡ 'I_1_2*up²') then

```

```

    xs_units0, i = 'cm6/s3',
    xs_mult0, i = const(1., -10)
else if (xs_var0, i ≡ 'I_2_4*up4) then
    xs_units0, i = 'cm7/s5,
    xs_mult0, i = const(1., -14)
else if (xs_var0, i ≡ 'sigv_max') then
    xs_units0, i = 'cm3/s'
    xs_mult0, i = const(1., -6)
else
    xs_units0, i = 'unknown'
@if 0
    xs_mult0, i = zero
@endif
    xs_mult0, i = one
end if
end do

/* Set defaults for unused dependent variables */
if (xs_num_dep_var < xs_dep_var_max) then
do j = xs_num_dep_var + 1, xs_dep_var_max

    xs_rankj = 1
    xs_eval_namej = 'unknown'
    xs_var0, j = 'unknown'
    xs_spacing0, j = 'unknown'
    xs_units0, j = 'unknown'
    xs_mult0, j = zero
    do i = 1, xs_table_rank_max
        xs_tab_indexi, j = 1
        xs_vari, j = 'unknown'
        xs_spacingi, j = 'unknown'
        xs_unitsi, j = 'unknown'
        xs_multi, j = zero
        xs_mini, j = zero
        xs_maxi, j = zero
    end do
end do
end if

return
end

```

Read in data for independent variables.

```

⟨ Functions and subroutines 2 ⟩ +≡
  subroutine read_indep(unit, line)
    implicit none f77
    xs_common
    implicit none f90
    integer unit      // Input
    character*(*) line      // Output
  @#if 0
    character*FILELEN tc      // Test String
  @#endiff
    integer length, p, b, e, i, idep, j      // Local
    integer dep_var xs_dep_var_max
    logical all_ints

    st_decls
    xs_decls

    do i = 1, xs_dep_var_max
      xs_rank_i = 0
    enddo

next_line: continue // Labels are defined in ratecalc.hweb
  assert(read_string(unit, line, length))
  if (line ≡ 'REACTIONS')
    goto end_indep_var
  assert(length ≤ len(line))
  length = parse_string(line(:length))

  p = 0
  assert(next_token(line(:length), b, e, p))
  /* We need to differentiate between the lines which list the indices of the dependent variables and
   those which enumerate the corresponding independent variables. The former should begin with
   a 1 or 2 digit integer; hence, the check on xs_dep_var_max. Set all_ints to true if these tokens
   are indeed integers. The independent variables are given by a multi-character string (all_ints =
   false). One in current use does begin with a "1"; hence, the care exercised here. */
  @#if 0
    tc = line(b:b)
    if (tc ≡ '0' ∨ tc ≡ '1' ∨ tc ≡ '2' ∨ tc ≡ '3' ∨ tc ≡ '4' ∨ tc ≡ '5' ∨ tc ≡ '6' ∨ tc ≡ '7' ∨ tc ≡
       '8' ∨ tc ≡ '9') then
  @#endiff
    if (b ≡ e) then
      all_ints = (lge(line(b:e), '1') ∧ lle(line(b:e), '9'))
    else if (e ≡ b + 1) then
      all_ints = (lge(line(b:e), '10') ∧ lle(line(b:e), '99'))
    else
      assert(xs_dep_var_max ≤ 99)
      all_ints = F
    end if
    if (all_ints) then
      idep = 1 // Read in list of dependent variables
      dep_var_1 = read_integer(line(b:e))
    end if
  end if
end subroutine

```

```

next_dep_var: continue
  if ( $\neg$ next_token(line(: length), b, e, p))
    goto next_line
    idep++
    dep_varidep = read_integer(line(b : e))
    goto next_dep_var

  else // And their corresponding independent variables
    xs_rankdep_var1++
    assert(xs_rankdep_var1 ≤ xs_table_rank_max)
    xs_varxs_rankdep_var1, dep_var1 = read_text(line(b : e))
    if (xs_varxs_rankdep_var1, dep_var1 ≡ 'none') then
      xs_rankdep_var1 = 0 // Constant
    else // Its minimum
      if (next_token(line(: length), b, e, p)) then
        xs_minxs_rankdep_var1, dep_var1 = read_real(line(b : e))
      else
        xs_minxs_rankdep_var1, dep_var1 = zero
    end if // Maximum
    if (next_token(line(: length), b, e, p)) then
      xs_maxxs_rankdep_var1, dep_var1 = read_real(line(b : e))
    else
      xs_maxxs_rankdep_var1, dep_var1 = zero
    end if // Number of values
    if (next_token(line(: length), b, e, p)) then
      xs_tab_indexxs_rankdep_var1, dep_var1 = read_integer(line(b : e))
    else
      if (xs_eval_namedep_var1 ≡ 'fit' ∧ xs_rankdep_var1 ≡ 1) then
        xs_tab_indexxs_rankdep_var1, dep_var1 = ra_fit_coef_max
      else
        xs_tab_indexxs_rankdep_var1, dep_var1 = 1
      end if
    end if // And spacing
    if (next_token(line(: length), b, e, p)) then
      xs_spacingxs_rankdep_var1, dep_var1 = read_text(line(b : e))
    else
      xs_spacingxs_rankdep_var1, dep_var1 = 'unknown'
    end if
  end if // Propagate to other
  if (idep > 1) then // dependent vars.
    do i = 2, idep
      xs_rankdep_vari = xs_rankdep_var1
      if (xs_rankdep_var1 > 0) then
        xs_varxs_rankdep_var1, dep_var1 = xs_varxs_rankdep_var1, dep_var1
        xs_minxs_rankdep_var1, dep_vari = xs_minxs_rankdep_var1, dep_var1
        xs_maxxs_rankdep_var1, dep_vari = xs_maxxs_rankdep_var1, dep_var1
        xs_tab_indexxs_rankdep_var1, dep_vari = xs_tab_indexxs_rankdep_var1, dep_var1
        xs_spacingxs_rankdep_var1, dep_vari = xs_spacingxs_rankdep_var1, dep_var1
      end if
    end do
  end if
end if

```

```

        end if
    end if
    goto next_line

end_indep_var: continue
    /* Assign cgs units with conversions to MKS */
    do j = 1, xs_num_dep_var
        if (xs_rank_j > 0) then
            do i = 1, xs_rank_j
                if (xs_var_i, j ≡ 'energy' ∨ xs_var_i, j ≡ 'temperature') then
                    xs_units_i, j = 'eV'
                    xs_mult_i, j = electron_charge
                else if (xs_var_i, j ≡ 'specific_energy' ∨ xs_var_i, j ≡ 'specific_temperature') then
                    xs_units_i, j = 'eV/amu'
                    xs_mult_i, j = electron_charge / atomic_mass_unit
                else if (xs_var_i, j ≡ 'density') then
                    xs_units_i, j = 'cm^-3'
                    xs_mult_i, j = const(1., 6)
                else
                    xs_units_i, j = 'unknown'
                @#if 0
                    xs_mult_i, j = zero
                @#endif
                    xs_mult_i, j = one
                end if
            end do
        end if    /* Fill defaults for unused ranks */
        if (xs_rank_j < xs_table_rank_max) then
            do i = xs_rank_j + 1, xs_table_rank_max
                xs_tab_index_i, j = 1
                xs_var_i, j = 'unknown'
                xs_spacing_i, j = 'unknown'
                xs_units_i, j = 'unknown'
                xs_mult_i, j = zero
                xs_min_i, j = zero
                xs_max_i, j = zero
            end do
        end if
    end do

    return
end

```

Read in data for reactions. This also forms the starting point for the rest of the ratecalc code which fills in the tables and writes the netCDF files for each reaction.

```
< Functions and subroutines 2 > +≡
subroutine read_reaction_list(unit, line)
  implicit none_f77
  xs_common
  ra_common
  implicit none_f90
  integer unit    // Input
  character(*) line    // Output
  integer length, p, b, e, i, idep    // Local
  integer dep_var xs_dep_var_max
  logical and_section

  st_decls
  xs_decls

  /* The following declarations are required to write out reaction specification information (ra class)
   as a netCDF file.
     integer fileid           character*FILELEN fileout
     nc_decls                 ra_ncdecl
```

```

*/
react_begin: continue // Labels are defined in ratecalc.hweb
    if (¬read_string(unit, line, length))
        line = 'END'
    if (line ≡ 'END')
        goto eof
    assert(length ≤ len(line))
    length = parse_string(line(: length))

    xs_name = line(: length) // First line: reaction name

    assert(read_string(unit, line, length)) // Second line: mass and units
    length = parse_string(line(: length))
    p = 0
    assert(next_token(line, b, e, p))
    ra_mass = read_real(line(b : e))
    assert(next_token(line, b, e, p))
    ra_mass_units = read_text(line(b : e))

    assert(read_string(unit, line, length)) // Third line: data filename
    length = parse_string(line(: length))
    assert(length ≤ len(line))
    ra_data_filename = line(: length)

    assert(read_string(unit, line, length)) // Fourth line: data version
    length = parse_string(line(: length))
    assert(length ≤ len(line))
    xsection_version = line(: length)

    assert(read_string(unit, line, length)) // Fifth line: hier. labels
    length = parse_string(line(: length))
    assert(length ≤ len(line))
    p = 0
    ra_num_hier = 0

next_hlab: continue
    if (¬next_token(line, b, e, p))
        goto bool_sect
    ra_num_hier++
    ra_hier_label[ra_num_hier] = read_text(line(b : e))
    goto next_hlab

    /* Subsequent pairs of lines provide the Boolean search labels for this reaction. */

bool_sect: continue
    do i = 1, xs_num_dep_var
        ra_num_bool_and[i] = 0
        ra_num_bool_not[i] = 0
    end do

new_bool_sect: continue // Start a new pair of lines
    /* If next line is blank or is the end of file, it indicates the end of this reaction specification. */
    idep = 0
    if (¬read_string(unit, line, length))
        goto react_done // End of file

```

```

length = parse_string(line(: length))
if (line(1 : 1) ≡ '-')
  goto react_done

p = 0
next_dep_var: continue // First line: dependent variables
if (¬next_token(line, b, e, p))
  goto bool_list
  idep++
  dep_var(idep) = read_integer(line(b : e))
  assert(dep_var(idep) ≤ xs_num_dep_var)
  goto next_dep_var

bool_list: continue
  assert(read_string(unit, line, length)) // Second: corresponding labels
  length = parse_string(line(: length))
  p = 0
  and_section = T

next_bool: continue
  if (¬next_token(line, b, e, p))
    goto new_bool_sect
    if (line(b : e) ≡ '&!')
      and_section = F
    if (and_section) then
      do i = 1, idep
        ra_num_bool_and_dep_var_i ++
        ra_bool_and_labelra_num_bool_and_dep_var_i, dep_var_i = read_text(line(b : e))
      end do
    else
      do i = 1, idep
        ra_num_bool_not_dep_var_i ++
        ra_bool_not_labelra_num_bool_not_dep_var_i, dep_var_i = read_text(line(b : e))
      end do
    end if
    goto next_bool
    /* Reaction specification complete */
react_done: continue
  /* The ra class data can be written to a netCDF file if needed:
     fileout='test.nc'
  fileid=nccreate(fileout,NC_CLOBBER,nc_stat)           ra_ncdef(fileid)
  call ncendef(fileid,nc_stat)                      ra_ncwrite(fileid)   call
  ncclose(fileid,nc_stat)
  Now fill tables and write data file */
  call fill_tables
  goto react_begin

eof: return
end

```

### 3 Fill Cross Section Tables

This routine carries out the loops over dependent and independent variables. Table values are obtained directly from ALADDIN (for fits) or via subroutine *table\_value*.

```

⟨Functions and subroutines 2⟩ +≡
subroutine fill_tables
  implicit none f77
  ra_common // Commons and declarations
  xs_common
  rf_common
  implicit none f90
  integer fileid, j, k, l, m, b, e, p // Local
  real zsigma, table_value
  real ind_var_min_xs_table_rank_max, ind_var_delta_xs_table_rank_max, ind_var_xs_table_rank_max
  character*FILELEN fileout
  character*LINELEN description

  xs_ncdecl
  nc_decls
  st_decls
  xs_decls
  ⟨Memory allocation interface 0⟩
  ra_localcommon // Local common
  xs_ragged_alloc(xs_data, xs_tab_index)

  do j = 1, xs_num_dep_var // Main loop
    /* Get fit information from ALADDIN. */
    if (xs_eval_namej ≠ 'table_external')
      call aladdin_read_coefs(ra_data_filename, ra_hier_label, ra_num_hier, ra_bool_and_label1,j,
                             ra_num_bool_andj, ra_bool_not_label1,j, ra_num_bool_notj, fit_coef, num_fit_coef, leval)
    if ((xs_eval_namej ≡ 'table_eval') ∨ (xs_eval_namej ≡ 'table_calc')) then
      /* For tables, find all delta values at this j */
      do k = 1, xs_table_rank_max
        call find_delta(xs_spacingk,j, xs_mink,j, one, xs_maxk,j, xs_tab_indexk,j, ind_var_mink,
                        ind_var_deltak)
      end do
      /* The following three loops represent the present size of xs_table_rank_max; if it is increased
         or decreased, the number of loops here should be altered accordingly. */
      do k = 0, xs_tab_index3,j - 1
        ind_var3 = ind_var_min3 + areal(k) * ind_var_delta3
        if (xs_spacing3,j ≡ 'log')
          ind_var3 = exp(ind_var3)
        do l = 0, xs_tab_index2,j - 1
          ind_var2 = ind_var_min2 + areal(l) * ind_var_delta2
          if (xs_spacing2,j ≡ 'log')
            ind_var2 = exp(ind_var2)
        do m = 0, xs_tab_index1,j - 1
          ind_var1 = ind_var_min1 + areal(m) * ind_var_delta1
          if (xs_spacing1,j ≡ 'log')

```

```

    ind_var1 = exp(ind_var1)
    zsigma = table_value(j, ind_var)
    xs_data_table(m, l, k, j) = zsigma
  end do
end do
/* End of "do" loops over xs_table indices

/* For fits, store name of evaluation function, number of fit coefficients, and the coefficients
   themselves. */
else if (xs_eval_namej ≡ 'fit') then
  xs_eval_namej = leval
  xs_tab_index1, j = num_fit_coef
  do m = 0, xs_tab_index1, j - 1
    xs_data_table(m, 0, 0, j) = fit_coefm+1
  end do
  /* For externally specified data, fill table with the appropriate number of zeroes. */
else if (xs_eval_namej ≡ 'table_external') then
  do k = 0, xs_tab_index3, j - 1
    do l = 0, xs_tab_index2, j - 1
      do m = 0, xs_tab_index1, j - 1
        xs_data_table(m, l, k, j) = zero
      end do
    end do
  end do
else
  assert('Unknown_organization' ≡ '_')
end if
/* End for "if" on xs_eval_name
end do
/* End of "do" over dependent variables

do j = xs_num_dep_var + 1, xs_dep_var_max
  do k = 0, xs_tab_index3, j - 1
    do l = 0, xs_tab_index2, j - 1
      do m = 0, xs_tab_index1, j - 1
        xs_data_table(m, l, k, j) = real_unused
      end do
    end do
  end do
end do
end do
/* All done. Write and close netCDF file */
p = 0
assert(next_token(xs_name, b, e, p))
fileout = xs_name(b : e) || '.nc'
fileid = nccreate(fileout, NC_CLOBBER, nc_stat)
description = 'Reference_atomic_physics_data_for_a_reaction_in_degas_2'
call ncattputc(fileid, NC_GLOBAL, 'description', NC_CHAR, string_length(description),
               description, nc_stat)

```

```
call ncattputc(fileid, NC_GLOBAL, 'data_version', NC_CHAR, string_length(xsection_version),
               xsection_version, nc_stat)

xs_ncdef(fileid)
call ncendef(fileid, nc_stat)
xs_ncwrite(fileid)

var_free(xs_data_tab)

call ncclose(fileid, nc_stat)

return
end
```

Table evaluation details. This routine uses the information about the dependent and independent variables to infer the type of evaluation which is required (cross section, electron reaction rate, ion reaction rate, etc.) and makes the appropriate function calls. The input parameters are  $j$  the index into the  $xs$  class variables for this dependent variable and  $ind\_var$  the values of the independent variables for this table entry.

```
<Functions and subroutines 2> +≡
function table_value(j, ind_var)
  implicit none_f77
  xs_common
  ra_common
  implicit none_f90

  real table_value // Function
  integer j // Input
  real ind_var_xs_table_rank_max

  integer num_fit_coef_aladdin // Local
  real norm_constant, infinite_integral, zsigma, zenergy, ztemperature
  double precision zte_aladdin, zdpsigma

  character*msg_length error

  external electron_rate_integrand, /* External */
           electron_norm_integrand, ion_rate_integrand
  real electron_rate_integrand, electron_norm_integrand, ion_rate_integrand

  ra_localcommon
  xs_decls /* For cross sections, presumably will only be evaluating fits to fill the table. Hence,
             moment_number is not needed. But, for higher moments, the variable moment_number must be
             set. Determine the required value from the contents of xs_var0, j. */
  if (xs_var0, j ≡ 'cross_section') then
    assert(xs_eval_namej ≡ 'table_eval')
  else if (xs_var0, j ≡ 'reaction_rate') then
    moment_number = 0
  else if (xs_var0, j ≡ 'I_1_0' ∨ xs_var0, j ≡ 'I_1_0') then
    moment_number = 0
  else if (xs_var0, j ≡ 'I_1_1*up' ∨ xs_var0, j ≡ 'I_0_1*up' ∨ xs_var0, j ≡ 'I_1_1*up') then
    moment_number = 1
  else if (xs_var0, j ≡ 'I_1_2*up2' ∨ xs_var0, j ≡ 'I_0_2*up2' ∨ xs_var0, j ≡ 'I_1_2*up2') then
    moment_number = 2
  else if (xs_var0, j ≡ 'I_2_4*up4') then
    moment_number = 4
  @#if 0
  else if (xs_var0, j ≡ 'sigv_max') then
    assert(xs_eval_namej ≡ 'table_external')
  else
    assert('Unknown_dependent_variable' ≡ '_')
  @#endif
  end if

  /* Now split up procedures according to rank of dependent variable. The sorts of things we
     expect to find with xs_rank = 1 are: cross sections (to be evaluated) and rates for electron
     impact processes. In the former case, the independent variable is an energy, temperature for
     the latter. */
  if (xs_rankj ≡ 1) then
    assert(xs_units1, j ≡ 'eV' ∨ xs_units1, j ≡ 'eV/amu')
```

```

if (xs_eval_namej ≡ 'table_calc') then
    assert (xs_var1,j ≡ 'temperature')
    temperature = ind_var1 // Calculate integral
    norm_constant = infinite_integral(electron_norm_integrand)
    num_evaluations = 0
    zsigma = infinite_integral(electron_rate_integrand)
    zsigma = zsigma / norm_constant

else // Evaluate fit
    if (xs_var1,j ≡ 'energy' ∨ xs_var1,j ≡ 'temperature') then
        zte_aladdin = dble(ind_var1) // It expects double
    else if (xs_var1,j ≡ 'specific_energy') then
        assert (ra_mass_units ≡ 'g')
        zte_aladdin = dble(ind_var1 * ra_mass / atomic_mass_unit_g)
    else
        assert ('Unexpected_value_of_xs_var[1,j]' ≡ ' ')
    end if
    zdpsigma = zero // Doesn't initialize
    call aladdin_eval_fit(leval, zte_aladdin, fit_coef, num_fit_coef, zdpsigma, error)
    zsigma = zdpsigma

end if

/* Alternatively, treat xs_rank = 2: expect an ion impact process. */
else if (xs_rankj ≡ 2) then
@#if 0
    // Move this below to make specific for table_calc only
    assert (xs_units1,j ≡ 'eV' ∨ xs_units1,j ≡ 'eV/amu')
    assert (xs_units2,j ≡ 'eV' ∨ xs_units2,j ≡ 'eV/amu')
    assert (ra_mass_units ≡ 'g')
    if ((xs_var1,j ≡ 'specific_energy') ∧ (xs_var2,j ≡ 'specific_temperature')) then
        zenergy = ind_var1
        ztemperature = ind_var2
    else if ((xs_var1,j ≡ 'specific_temperature') ∧ (xs_var2,j ≡ 'specific_energy')) then
        ztemperature = ind_var1
        zenergy = ind_var2
    else
        assert ('Unexpected_values_of_xs_var' ≡ ' ')
    end if
@#endif
if (xs_eval_namej ≡ 'table_calc') then
    assert (xs_units1,j ≡ 'eV' ∨ xs_units1,j ≡ 'eV/amu')
    assert (xs_units2,j ≡ 'eV' ∨ xs_units2,j ≡ 'eV/amu')
    assert (ra_mass_units ≡ 'g')
    if ((xs_var1,j ≡ 'specific_energy') ∧ (xs_var2,j ≡ 'specific_temperature')) then
        zenergy = ind_var1
        ztemperature = ind_var2
    else if ((xs_var1,j ≡ 'specific_temperature') ∧ (xs_var2,j ≡ 'specific_energy')) then
        ztemperature = ind_var1
        zenergy = ind_var2
    else
        assert ('Unexpected_values_of_xs_var' ≡ ' ')
    end if
    up = sqrt(two * ztemperature)

```

```

    u = sqrt(two * zenergy)
    num_evaluations = 0
    zsigma = infinite_integral(ion_rate_integrand)
else
@if 0
    // Try to generalize here, but add a specific check for JANRD where we know what the order
    // of the arguments should be.
    zte_aladdin = dble(ztemperature)
#endifendif
    assert(¬((leval ≡ '#JANRD') ∧ (xs_var1, j ≡ 'specific_energy') ∧ (xs_var2, j ≡
        'specific_temperature')))
    if (xs_var1, j ≡ 'specific_temperature' ∨ xs_var1, j ≡ 'specific_energy') then
        assert(ra_mass_units ≡ 'g')
        zte_aladdin = dble(ind_var1 * ra_mass / atomic_mass_unit_g)
    else
        zte_aladdin = dble(ind_var1)
    end if
    num_fit_coef_aladdin = num_fit_coef + 1
#endifif 0
    fit_coef_num_fit_coef_aladdin = dble(zenergy)
#endifendif
    if (xs_var2, j ≡ 'specific_temperature' ∨ xs_var2, j ≡ 'specific_energy') then
        assert(ra_mass_units ≡ 'g')
        fit_coef_num_fit_coef_aladdin = dble(ind_var2 * ra_mass / atomic_mass_unit_g)
    else
        fit_coef_num_fit_coef_aladdin = dble(ind_var2)
    end if
    zdpsigma = zero
    call aladdin_eval_fit(leval, zte_aladdin, fit_coef, num_fit_coef_aladdin, zdpsigma, error)
    zsigma = areal(zdpsigma)
end if
else
    assert('Ranks higher than 2 not treated' ≡ ' ')
end if
table_value = zsigma
return
end

```

Compute Dependent Variable “delta”s. These steps will be executed frequently throughout the code. Perhaps this routine can be standardized and used in many places. The minimum *min\_value* and maximum *max\_value* are input. A uniform series of length *number\_of\_values*, either linear or logarithmic according to *spacing*, is to be established with the spacing *var\_delta*. At the same time, the original values are to be rescaled by *multiplier*; typically, this is used to effect a change in units. Hence, the output minimum *var\_min* may differ in general from the input *min\_value*.

```

⟨ Functions and subroutines 2 ⟩ +≡
subroutine find_delta(spacing, min_value, multiplier, max_value, number_of_values, var_min,
    var_delta)
implicit none_f77
implicit none_f90
real min_value, multiplier, max_value // Input
integer number_of_values
character*xs_tag_string_length spacing
real var_min, var_delta // Output

if (number_of_values ≤ 1) then
    var_min = zero
    var_delta = zero
    return
end if

if (spacing ≡ 'linear') then
    var_min = min_value * multiplier
    var_delta = (max_value - min_value) * multiplier / areal(number_of_values - 1)
else if (spacing ≡ 'log') then
    var_min = log(min_value * multiplier)
    var_delta = log(max_value / min_value) / areal(number_of_values - 1)
else
    assert('Unknown_spacing_type' ≡ '_')
end if

return
end

```

## 4 Integrands for Rate Calculations

This routine is a single argument function which provides the integrand in a calculation of velocity-weighted averages of a user-specified function over a user-specified distribution function. The principle use of this routine is to compute a velocity-weighted reaction rate for electrons colliding with neutral particles,

$$\langle \sigma v v^n \rangle = \frac{4\pi}{\alpha m} \left( \frac{2}{m} \right)^{(n+2)/2} \int_0^\infty dE \sigma f(E) E^{(n+2)/2}, \quad (4)$$

where  $\sigma(E)$  is computed using the function *sigma*,  $f(E)$  is calculated by the function *e\_distrib\_fn*,  $m$  is the particle mass relating  $E$  and  $v$  via  $E = \frac{1}{2}mv^2$ , and  $\alpha$  is the normalization constant for the distribution function,

$$\alpha = \frac{4\pi}{m} \left( \frac{2}{m} \right)^{1/2} \int_0^\infty dE f(E) E^{1/2}.$$

Note that  $\alpha$  does not appear in this routine but is incorporated in the calling routines.

This routine is set up with a single argument so that it can be used with “canned” integration routines; the argument *energy* is the energy  $E$  in the above expressions. In order for the numerical integrations to work effectively, it is necessary that  $E$  be in practical units; eV in this case. Appropriate unit conversions have been inserted into the overall multiplier to put the result in cgs units. The mass  $m = ra\_mass$  and exponent  $n = moment\_number$  must come in through common blocks.

```
< Functions and subroutines 2 > +≡
function electron_rate_integrand(energy)
  implicit none_f77
  ra_common // Common
  implicit none_f90
  real electron_rate_integrand
  real energy // Input
  real multiplier, ze_dist, exponent, zint, zsigma // Local variables
  real e_distrib_fn, sigma // External functions
  ra_localcommon // Local common
  zsigma = sigma(energy)
  ze_dist = e_distrib_fn(energy)
  exponent = half * (areal(moment_number) + two)
  multiplier = (const(4.0) * PI / ra_mass) * (two / ra_mass)exponent * ev_to_ergsexponent+one
  zint = multiplier * zsigma * ze_dist * energyexponent
  electron_rate_integrand = zint
return
end
```

This routine is the integrand used to compute the normalization constant for electron reaction rates. The primary motivation for doing this numerically is allow the use of arbitrary electron distribution functions. The argument *energy* is assumed to be in eV; the overall multiplier converts the resulting units to cgs.

```
<Functions and subroutines 2> +≡
function electron_norm_integrand(energy)
  implicit none_f77
  ra_common // Common
  implicit none_f90
  real electron_norm_integrand
  real energy // Input

  real multiplier, ze_dist, zint // Local variables
  real e_distrib_fn // External functions

  ze_dist = e_distrib_fn(energy)
  multiplier = (const(4.0) * PI / ra_mass) * sqrt(two / ra_mass) * ev_to_ergs const(1.5)
  zint = multiplier * ze_dist * sqrt(energy)

  electron_norm_integrand = zint

  return
end
```

This routine is a single argument function which provides the integrand in a calculation velocity-weighted reaction rates for cases involving two “heavy” particles. The plasma species is assumed to be described by a Maxwellian distribution function. The desired integral can be expressed in terms of[1]

$$I_{\ell,n} \equiv \frac{1}{\sqrt{\pi} u u_p^{n+1}} \int_0^\infty dv_{rel} v_{rel}^{2+n} \sigma_\ell(v_{rel}) \left\{ \exp \left[ \frac{-(v_{rel} - u)^2}{u_p^2} \right] - (-1)^n \exp \left[ \frac{-(v_{rel} + u)^2}{u_p^2} \right] \right\}, \quad (5)$$

where  $u$  is the velocity of the neutral particle, and  $u_p$  is the thermal velocity of the plasma particle. The latter is related to the plasma temperature by  $T_p = \frac{1}{2} m_p u_p^2$ . The result provided by this routine is  $I_{\ell,n} u_p^n$  and, thus, has the same dimensions as  $\langle \sigma v v^n \rangle$ .

This routine is set up with a single argument so that it can be used with “canned” integration routines; the argument *rel\_velocity* is the relative velocity  $v_{rel}$  in the above expressions. The factors  $u$  and  $u_p$ , as well as the exponent  $n$  is represented by the variable *moment\_number*. Also in common is the mass *ra\_mass* used to convert the relative velocity into an energy for use in evaluating the cross section *sigma*. The value to be used for *ra\_mass* may vary, depending on the origin of the data used in *sigma*.

The units of the various velocities are taken to be  $\sqrt{\text{eV}/\text{amu}}$  so that values of order unity for *rel\_velocity* are reasonable. It is assumed that *ra\_mass* has units of grams, that the argument of *sigma* should be in eV, and that *sigma* itself is in  $\text{cm}^2$ . The factor *multiplier* is defined so that the resulting integral has cgs units.

```
<Functions and subroutines 2> +≡
function ion_rate_integrand(rel_velocity)
  implicit none_f77
  ra_common // Common
  implicit none_f90
  real ion_rate_integrand
  real rel_velocity // Input
  real zbracket, multiplier, ze, exponent_minus, /* Local */
        exponent_plus, zsigma
  real sigma // External function
  ra_localcommon // Local common
  ze = half * (ra_mass / atomic_mass_unit_g) * rel_velocity**2
  zsigma = sigma(ze)
  exponent_plus = -(rel_velocity + u)**2 / up**2
  exponent_minus = -(rel_velocity - u)**2 / up**2
  zbracket = exp(exponent_minus) - (-1)**moment_number * exp(exponent_plus)
  multiplier = one / (sqrt(PI) * u * up) * (sqrt(ev_to_ergs / atomic_mass_unit_g))**1+moment_number
  ion_rate_integrand = multiplier * rel_velocity**2+moment_number * zsigma * zbracket
  return
end
```

This routine provides the electron distribution function at the energy *energy*. Presently, the distribution is assumed to be Maxwellian; the temperature *temperature* characterizing the distribution is passed in through common. Note that *temperature* and *energy* must have the same units. The routine utilizes a single argument for generality since other distributions may require additional parameters (beyond temperature) to describe them.

```
< Functions and subroutines 2 > +≡
function e_distrib_fn(energy)
  implicit none_f77
  implicit none_f90
  real e_distrib_fn
  real energy // Input
  ra_localcommon // Local common
  e_distrib_fn = exp(-energy / temperature)
  return
end
```

This routine serves as the intermediary between the integration functions and calls to ALADDIN subroutines. In order to fit in with the former, this function must have only a single argument; consequently, it can easily be altered without having to change the calling routines.

As presently used, this routine computes the cross section at the input energy *energy*. The other information required to communicate with ALADDIN is brought in by the common block *ra\_localcommon*.

*<Functions and subroutines 2> +≡*

```

function sigma(energy)
  implicit none_f77
  implicit none_f90
  real sigma
  real energy // Input

  double precision ze, zfit // Local
  character*msg_length errmsg

  ra_localcommon // Local common

  ze = dble(energy) // Explicitly transfer to double precision
  errmsg = 'U'
  num_evaluations = num_evaluations + 1
  call aladdin_eval_fit(leval, ze, fit_coef, num_fit_coef, zfit, errmsg)

  /* Trap errors from ALADDIN routines. Barring actual foul-ups, the the only likely error is from
   requesting cross section values at energies below threshold. A handful of the ALADDIN routines
   report this as an error condition. Assume this is the case here. */
  if (errmsg ≠ 'U') then
    zfit = 0.0 · 100D
    /* write(stderr,'Warning from sigma and aladdin: ',errmsg)*/
  end if

  sigma = areal(zfit) // Explicitly convert back to normal precision

  return
end

```

## 5 ALADDIN Related Routines

This one reads fit coefficients. It takes as input *filename*, the name of an ALADDIN data file; *hier\_label*, an array of *num\_hier* hierarchical search labels; *bool\_and\_label*, an array of *num\_bool\_and* “.and.” boolean search labels; and *bool\_not\_label*, an array of *num\_bool\_not* “.not.” boolean search labels (see ALADDIN documentation for further details). Following the search through *filename*, the fit *num\_fit\_coef* fit coefficients are output in the array *cf*. The boolean label identifying the evaluation routine is returned in *leval*.

```
<Functions and subroutines 2> +≡
subroutine aladdin_read_coefs(filename, hier_label, num_hier, bool_and_label, num_bool_and,
    bool_not_label, num_bool_not, fit_coef, num_fit_coef, leval)
implicit none_f77
implicit none_f90
character*ra_data_filename_length filename // Input
character*ra_search_label_length hier_label(ra_label_max), bool_and_label(ra_label_max),
    bool_not_label(ra_label_max)
integer num_hier, num_bool_and, num_bool_not
character*xs_eval_name_length leval // Output
integer num_fit_coef
double precision fit_coef(ra_fit_coef_max)
decl_alpcom // Need bl and nbl from here
logical end_of_file, match // Local variables
character*msg_length errormsg
integer ientry, alad_diskin
alad_diskin = diskin + 1
open(alad_diskin, file = filename, status = 'old')
/* Loop through entries in filename */
ientry = 0
loop: continue
    call already(alad_diskin, ientry, end_of_file, errormsg)
    assert( $\neg$ end_of_file)
    assert(errormsg  $\equiv$  'u')
    /* Compare this entry with labels provided on input */
    call alcomp(hier_label, num_hier, bool_and_label, num_bool_and, bool_not_label, num_bool_not, match)
    if ( $\neg$ match)
        goto loop
    close(unit = alad_diskin) // Have a match
    /* Decode coefficients; check first that access label is the standard one */
    assert(bl_1SP  $\equiv$  '$u')
    call alrecf(fit_coef, num_fit_coef, ra_fit_coef_max, errormsg)
    assert(errormsg  $\equiv$  'u')
    leval = bl_nbl // Assign evaluation label variable
    /* Note: this step is not really necessary; could just return bl and select the last value when
       calling the evaluation routine. */
    return
end
```

Evaluate ALADDIN fits. This routine does the dirty job of associating the input boolean label for the evaluation routine, *leval*, with an actual FORTRAN subroutine. Because all of the ALADDIN routines of interest currently have essentially the same arguments (although they differ in details which are of no consequence to the compiler), this clearing-house routine can be generated automatically from a shell script which reads through the file containing all of the ALADDIN subroutines.

In addition to *leval*, the input arguments are: *pet*, the energy or temperature at which the ALADDIN fit is to be evaluated and *fit\_coef*, the array of *num\_fit\_coef* fit coefficients passed in from the ALADDIN data files. The resulting value is returned as *pfit*. Errors are returned via *kernmsg*.

```

⟨ Functions and subroutines 2 ⟩ +≡
subroutine aladdin_eval_fit(leval, pet, fit_coef, num_fit_coef, pfit, kernmsg) implicit_none_f77
implicit_none_f90

character*xs_eval_name_length leval // Input
integer num_fit_coef
double precision pet
double precision fit_coef(ra_fit_coef_max)

double precision pfit // Output
character*msg_length kernmsg

if (leval ≡ '#ALMEWE') then
    call ALMEWE(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#BELI') then
    call BELI(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#ALCHEB') then
    call ALCHEB(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#ALKING') then
    call ALKING(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#ALPHACX') then
    call ALPHACX(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#JAN1') then
    call JAN1(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#ETABSQ') then
    call ETABSQ(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#FNAGEX') then
    call FNAGEX(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#FNRC') then
    call FNRC(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#FNXS') then
    call FNXS(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#FORC') then
    call FORC(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#FOXS') then
    call FOXS(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#JBORN1') then
    call JBORN1(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#JBORN2') then
    call JBORN2(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#JANRD') then
    call JANRD(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#EEXCH2') then
    call EEXCH2(pet, fit_coef, num_fit_coef, pfit, kernmsg)
else if (leval ≡ '#EIONH2') then

```

```

call EIONH2(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#EIONHR') then
  call EIONHR(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JRREC1') then
  call JRREC1(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JRREC3') then
  call JRREC3(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JEEXC1') then
  call JEEXC1(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JEEXC2') then
  call JEEXC2(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JEEXC3') then
  call JEEXC3(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JEEXC4') then
  call JEEXC4(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JEEXC6') then
  call JEEXC6(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JIONHE') then
  call JIONHE(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JEION5') then
  call JEION5(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JRREC2') then
  call JRREC2(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JEEXC5') then
  call JEEXC5(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JEEXC7') then
  call JEEXC7(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JDREC1') then
  call JDREC1(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#ALJAN3') then
  call ALJAN3(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#PEXCH4') then
  call PEXCH4(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#IONBEA') then
  call IONBEA(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JCX1') then
  call JCX1(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#JCX2') then
  call JCX2(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#NEEXH1') then
  call NEEXH1(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#NEEXH2') then
  call NEEXH2(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#EEXCJN') then
  call EEXCJN(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#EIONJN') then
  call EIONJN(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#HEXC1') then
  call HEXC1(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#HEXC2') then
  call HEXC2(pet, fit_coef, num_fit_coef, pfit, kermmsg)
else if (leval ≡ '#HEXC3') then

```

```

call HEXC3(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#HEXC4') then
  call HEXC4(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#HEXC5') then
  call HEXC5(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#HEXCLD') then
  call HEXCLD(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#HIONN') then
  call HIONN(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#HCX1') then
  call HCX1(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#HCX2') then
  call HCX2(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#HCX3') then
  call HCX3(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#AQEXC1') then
  call AQEXC1(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#AQEXC2') then
  call AQEXC2(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#AQEXC3') then
  call AQEXC3(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#AIONGL') then
  call AIONGL(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#HCX4') then
  call HCX4(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#ACXGL1') then
  call ACXGL1(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#ACXGL2') then
  call ACXGL2(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#REFL1') then
  call REFL1(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#REFL2') then
  call REFL2(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#REFL3') then
  call REFL3(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#SPTEK') then
  call SPTEK(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#SPTTH') then
  call SPTTH(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#ELCROSS') then
  call ELCROSS(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#ELSPLINE') then
  call ELSPLINE(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#INTERP2D') then /* INTERP2D and INTERP1D, like other Aladdin routines
                                explicitly use double precision. However, for convenience they make use of the find_delta
                                and interpolation routines in interpolate.web. This assert will alert the user to a precision
                                incompatibility with those routines. */
  assert ( index ( $STRING ( real ) , 'DOUBLE' ) > 0 )
  call INTERP2D(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
else if (leval ≡ '#INTERP1D') then assert ( index ( $STRING ( real ) , 'DOUBLE' ) > 0 )
  call INTERP1D(pet, fit_coef, num_fit_coef, pfit, kerrmsg)
end if

```

```
return end
```

## 6 Integration Routines

This first function is used to hide some of the machinery needed to compute the infinite integrals required by this code. It is assumed that all integrals have limits  $0 \rightarrow \infty$ . In practice, however, the integrand usually exhibits some sort of threshold behavior so that the actual lower limit is nonzero. The integrands typically vanish exponentially for large values of the variable of integration. Hence, there is also a practical upper limit to the integral where the integrand vanishes numerically. In addition, the integrand may utilize a fit function of some sort and take a zero value outside the range of validity of that fit. The upshot is again that the range of integration is effectively smaller than  $0 \rightarrow \infty$ . Since these discontinuities, particularly the last type, can cause undue strain on the integration routine, it is worthwhile to explicitly locate these “thresholds” and modify the limits of integration accordingly. Additional function evaluations are used, perhaps more than the integrator would otherwise need, but the result is a more robust procedure.

Even with a finite range of integration, a variable transformation may be used to shrink the range to fit within the interval  $0 \rightarrow 1$ . For the transformation used here, this is difficult to do with a single transformation if the lower limit is still  $< 1$ . Consequently, the interval may be split into two parts so that separate transformations can be used on each.

**Note:** The choice of *dcadre* as the integration routine used here represents a compromise. On the one hand, the routine needs to be portable and easy to distribute (e.g., this requirement rules out most commercial packages). On the other hand, the method must be robust and relatively efficient. Because discontinuities in the sorts of integrands to be used here are not unusual (particularly in the first derivative), simple high order schemes fail miserably. While a trapezoidal rule works admirably in these cases, it is terribly slow, requiring many millions of function evaluations to achieve a reasonable accuracy. A proper treatment of these situations requires an adaptive routine; the literature indicates that an efficient implementation of an adaptive scheme is tricky. This led to the identification of *dcadre* as a candidate routine, a public-domain routine that has been used previously in commercial libraries.

Even then, it was decided that the sudden onsets at thresholds and fit limits was causing *dcadre* to struggle excessively. For this reason, the *find\_threshold* procedure was implemented. The use of a variable transformation in combination with finite limits appeared to further improve the efficiency of the integrations.

If an integral fails to converge, the user can try the following:

1. Increase the error tolerance, *integration\_acc*.
2. Try turning off the variable transformation; i.e., set *integ\_transform\_ratio* = 0.
3. Eliminate the threshold calculation (replace bounds in calls to *dcadre* with the original ones).
4. Get a better integration routine! (Better ones may exist).

```

⟨ Functions and subroutines 2 ⟩ +≡
function infinite_integral(integrand)
  implicit_none_f77
  implicit_none_f90
  real infinite_integral
  integer ier // Local
  real deadre, lower_result, upper_result, error, lower_bound_rev, upper_bound_rev
  external integrand // External
  real integrand

  /* Locate “thresholds” if any and alter bounds accordingly; the accuracy used here is the same as
   that used for the integrator. */

  call find_threshold(zero, integration_infinity, integrand, integration_acc, lower_bound_rev,
    upper_bound_rev)

  /* If lower limit is still < 1 (e.g., possibly still zero), split integral into two pieces so that a
   variable transformation can be used on the upper half if needed. */
  if (lower_bound_rev < one) then
    lower_result = dcadre(integrand, lower_bound_rev, one, zero, integration_acc, error, ier)
    assert(ier < 100)
    upper_result = dcadre(integrand, one, upper_bound_rev, zero, integration_acc, error, ier)
    assert(ier < 100)
  else
    lower_result = zero
    upper_result = dcadre(integrand, lower_bound_rev, upper_bound_rev, zero, integration_acc,
      error, ier)
    assert(ier < 100)
  end if

  infinite_integral = lower_result + upper_result

  return
end

```

Cautious adaptive Romberg extrapolation integrator. This routine computes  $\int_a^b f(x)dx$  using a cautious adaptive Romberg extrapolation. The user provides  $f$ , a single-argument real function subprogram; it must be declared external in the calling program.  $aerr$  is the desired *absolute* error in the answer;  $rerr$  is the desired *relative* error in the answer. On output  $dcadre$  returns  $error$ , an estimated bound on the absolute error of the integral. Non-zero values of the error parameter  $ier$  indicate one of the following conditions:

- Warning error with  $ier = 64 + n$ ,
  - $n = 1$  implies that one or more singularities were successfully handled.
  - $n = 2$  implies that, in some subinterval(s), the estimate of the integral was accepted merely because the estimated error was small, even though no regular behavior was recognized.
- Terminal error with  $ier = 128 + n$ ,
  - $n = 3$ , failure due to insufficient internal working storage.
  - $n = 4$ , failure; may be due to too much noise in the function (relative to the given error requirements) or due to an ill-behaved integrand.
  - $n = 5$  indicates that  $rerr > 0.1$ ,  $rerr < 0$ , or  $rerr$  is too small for the precision of the machine.

This routine is similar to one found in commercial numerical software packages. References [4] and [5] describe this routine and provide text versions of the source code. It is assumed that this routine is in the public domain. The code has been altered slightly to make it compatible with the rest of this (WEB) program. Otherwise, the “latest revision” date of January 27, 1977 is still applicable.

```
<Functions and subroutines 2> +≡
function dcadre(f, lower_bound, upper_bound, aerr, rerr, error, ier)
  implicit none_f77
  implicit none_f90
  real dcadre
  real lower_bound, upper_bound, f, aerr, rerr // Input
  integer ier // Output
  real error

  integer i, ibeg, iend, ii, iii, istage, istep, istep2, /* Local */
  it, l, lm1, maxtbl, maxsts, mxstge, n, n2, nnleft
  real a, absi, aitlow, aittol, alg4o2, alpha, astep, b, beg, cadre, curest, diff, edn, ergr, ergoal, erra,
       ererr, erret, errr, f_sub, fbeg, fbeg2, fend, fextm1, fextrp, fi, fn, fnsiz, four, fourp5, h2next,
       h2tfex, h2tol, hovn, hun, jumptl, length, p1, prever, sing, singnx, slope, stage, step, stepmn,
       steppnm, sum, sumabs, tabs, tabtlm, ten, x, vint
  real t10, 10, r10, ait10, dif10, rn4, ts2049, ibegs30, begin30, finis30, est30
  logical h2conv, inf_integ, aitken, right, reglar
  logical reglsv30

  data maxsts, maxtbl, mxstge/2049, 10, 30/

f_sub(x) = f(x-integ_transform_exp) * (areal(integ_transform_exp) / xinteg_transform_exp+1)

/* In the original version the following variables were assigned in data statements. We use the
   “const” macro here to ensure that proper precision is used. */

aitlow = const(1.1)
h2tol = const(0.15)
aittol = const(0.1)
jumptl = const(0.01)
rn1 = const(0.7142005)
rn2 = const(0.3466282)
rn3 = const(0.843751)
rn4 = const(0.1263305)
p1 = const(0.1)
four = const(4.0)
fourp5 = const(4.5)
ten = const(10.0)
hun = const(100.0)

alg4o2 = log10(two) // Initializations
cadre = zero
error = zero
curest = zero
vint = zero
ier = 0

if (max(one, lower_bound) / upper_bound < integ_transform_ratio) then
  a = upper_bound - one / areal(integ_transform_exp)
  b = lower_bound - one / areal(integ_transform_exp)
  inf_integ = T
```

```

else
  a = lower_bound
  b = upper_bound
  inf_integ =  $\mathcal{F}$ 
end if

length = abs(b - a)
if (length ≡ zero)
  go to 215
if (rerr > p1 ∨ rerr < zero)
  go to 210
if (aerr ≡ zero ∧ (rerr + hun) ≤ hun)
  go to 210
errr = rerr
erra = abs(aerr)
stepmn = (length / areal(2mxstge))
stepnm = max(length, abs(a), abs(b)) * ten
stage = half
istage = 1
fnsize = zero
prever = zero
reglar =  $\mathcal{F}$ 

/* The given interval of integration is the first interval considered. */
beg = a
if (inf_integ) then
  fbeg = f_sub(beg) * half
else
  fbeg = f(beg) * half
end if
ts1 = fbeg
ibeg = 1
edn = b
if (inf_integ) then
  fend = f_sub(edn) * half
else
  fend = f(edn) * half
end if
ts2 = fend
iend = 2
5: right =  $\mathcal{F}$ 

/* Investigation of a particular subinterval begins at this point. */
10: step = edn - beg
  astep = abs(step)
  if (astep < stepmn)
    go to 205
  if (stepnm + astep ≡ stepnm)
    go to 205
  t1,1 = fbeg + fend
  tabs = abs(fbeg) + abs(fend)
  l = 1
  n = 1
  h2conv =  $\mathcal{F}$ 

```

```

 $aitken = \mathcal{F}$ 
15:  $lm1 = l$ 
 $l = l + 1$ 

/* Calculate the next trapezoid sum,  $t_{l, 1}$ , which is based on  $n2 + 1$  equispaced points. Here,
 $n2 = 2n = 2^{l-1}$ . */
 $n2 = n + n$ 
 $fn = areal(n2)$ 
 $istep = (iend - ibeg) / n$ 
if ( $istep > 1$ )
  go to 25
 $ii = iend$ 
 $iend = iend + n$ 
if ( $iend > maxts$ )
  go to 200
 $hovn = step / fn$ 
 $iii = iend$ 
 $fi = one$ 
do  $i = 1, n2, 2$ 
   $ts_{iii} = ts_{ii}$ 
  if ( $inf\_integ$ ) then
     $ts_{iii-1} = f\_sub(edn - fi * hovn)$ 
  else
     $ts_{iii-1} = f(edn - fi * hovn)$ 
  end if
   $fi = fi + two$ 
   $iii = iii - 2$ 
   $ii = ii - 1$ 
end do
 $istep = 2$ 
25:  $istep2 = ibeg + istep / 2$ 
 $sum = zero$ 
 $sumabs = zero$ 
do  $i = istep2, iend, istep$ 
   $sum = sum + ts_i$ 
   $sumabs = sumabs + abs(ts_i)$ 
end do
 $tl_{l, 1} = tl_{l-1, 1} * half + sum / fn$ 
 $tabs = tabs * half + sumabs / fn$ 
 $absi = astep * tabs$ 
 $n = n2$ 

/* Get preliminary value for  $vint$  from last trapezoid sum and update the error requirement  $ergoal$ 
   for this subinterval. */
 $it = 1$ 
 $vint = step * tl_{l, 1}$ 
 $tabtlm = tabs * ten$ 
 $fnsize = max(fnsize, abs(tl_{l, 1}))$ 
 $ergl = astep * fnsize * ten$ 
 $ergoal = stage * max(erra, errr * abs(curest + vint))$ 

/* Complete row  $l$  and column  $l$  of  $t$  array. */
 $fextrp = one$ 
do  $i = 1, lm1$ 

```

```

fextrp = fextrp * four
ti, l = tl, i - tl-1, i
tl, i+1 = tl, i + ti, l / (fextrp - one)
end do
error = astep * abs(t1, l)
/* Preliminary decision procedure: if l = 2 and t2, 1 = t1, 1, go to 135 to follow up the impression
   that intergrand is a straight line. */
if (l > 2)
   go to 40
if (tabs + p1 * abs(t1, 2) ≡ tabs)
   go to 135
go to 15
/* Calculate next ratios for columns 1,...,l-2 of t-table. Ratio is set to zero if difference in last two
   entries of column is about zero. */
40: do i = 2, lm1
    diff = zero
    if (tabtlm + abs(ti-1, l) ≠ tabtlm)
       diff = ti-1, lm1 / ti-1, l
       ti-1, lm1 = diff
    end do
    if (abs(four - t1, lm1) ≤ h2tol)
       go to 60
    if (t1, lm1 ≡ zero)
       go to 55
    if (abs(two - abs(t1, lm1)) < jumpth)
       go to 130
    if (l ≡ 3)
       go to 15
    h2conv = F
    if (abs((t1, lm1 - t1, l-2) / t1, lm1) ≤ aittol)
       go to 75
50: if (reglar)
    go to 55
    if (l ≡ 4)
       go to 15
55: if (error > ergoal ∧ (ergl + error) ≠ ergl)
    go to 175
    go to 145 /* Cautious Romberg extrapolation */
60: if (h2conv)
    go to 65
    aitken = F
    h2conv = T
65: fextrp = four
70: it = it + 1
    vint = step * tl, it
    error = abs(step / (fextrp - one) * tit-1, l)
    if (error ≤ ergoal)
       go to 160
    if (ergl + error ≡ ergl)
       go to 160
    if (it ≡ lm1)

```

```

        go to 125
    if ( $t_{it, lm1} \equiv zero$ )
        go to 70
    if ( $t_{it, lm1} \leq fextrp$ )
        go to 125
    if ( $\text{abs}(t_{it, lm1} / four - fextrp) / fextrp < aittol$ )
         $fextrp = fextrp * four$ 
    go to 70

    /* Integrand may have  $x^\alpha$  type singularity resulting in a ratio of  $sing = 2^{\alpha+1}$ . */
75: if ( $t_{1, lm1} < aitlow$ )
    go to 175
    if (aitken)
        go to 80
     $h2conv = \mathcal{F}$ 
    aitken =  $\mathcal{T}$ 
80:  $fextrp = t_{l-2, lm1}$ 
    if ( $fextrp > fourp5$ )
        go to 65
    if ( $fextrp < aitlow$ )
        go to 175
    if ( $\text{abs}(fextrp - t_{l-3, lm1}) / t_{1, lm1} > h2tol$ )
        go to 175
     $sing = fextrp$ 
     $fextm1 = one / (fextrp - one)$ 
     $ait_1 = zero$ 
    do  $i = 2, l$ 
         $ait_i = t_{i, 1} + (t_{i, 1} - t_{i-1, 1}) * fextm1$ 
         $r_i = t_{1, i-1}$ 
         $dif_i = ait_i - ait_{i-1}$ 
    end do
     $it = 2$ 
90:  $vint = step * ait_l$ 
     $errel = errel * fextm1$ 
    if ( $errel > ergoal \wedge (ergl + errel) \neq ergl$ )
        go to 95
     $alpha = \log_{10}(sing) / alg4o2 - one$ 
     $ier = \max(ier, 65)$ 
    go to 160
95:  $it = it + 1$ 
    if ( $it \equiv lm1$ )
        go to 125
    if ( $it > 3$ )
        go to 100
     $h2next = four$ 
     $singnx = sing + sing$ 
100: if ( $h2next < singnx$ )
    go to 105
     $fextrp = singnx$ 
     $singnx = singnx + singnx$ 
    go to 110
105:  $fextrp = h2next$ 
     $h2next = four * h2next$ 

```

```

110: do  $i = it, lm1$ 
       $r_{i+1} = zero$ 
      if ( $tabtlm + \text{abs}(dif_{i+1}) \neq tabtlm$ )
           $r_{i+1} = dif_i / dif_{i+1}$ 
      end do
       $h2tfex = -h2tol * fextrp$ 
      if ( $r_l - fextrp < h2tfex$ )
          go to 125
      if ( $r_{l-1} - fextrp < h2tfex$ )
          go to 125
       $ererr = astep * \text{abs}(dif_l)$ 
       $fextm1 = one / (fextrp - one)$ 
      do  $i = it, l$ 
           $ait_i = ait_i + dif_i * fextm1$ 
           $dif_i = ait_i - ait_{i-1}$ 
      end do
      go to 90

      /* Current trapezoid sum and resulting extrapolated values did not give a small enough  $ererr$ .
         Note, having  $prever < ererr$  is an almost certain sign of beginning trouble with in the function
         values. Hence, a watch for, and control of, noise should begin here. */

125:  $fextrp = \max(prever / ererr, aitlow)$ 
     $prever = ererr$ 
    if ( $l < 5$ )
        go to 15
    if ( $l - it > 2 \wedge istage < mxstge$ )
        go to 170
     $erret = ererr / (fextrp^{maxtbl-l})$ 
    if ( $erret > ergoal \wedge (ergl + erret) \neq ergl$ )
        go to 170
    go to 15

    /* Integrand has jump */
130: if ( $erret > ergoal \wedge (ergl + erret) \neq ergl$ )
    go to 170

     $diff = \text{abs}(t_{1,l}) * (fn + fn)$  // Note that  $2fn = 2^l$ 
    go to 160

    /* The integrand is a straight line: test this assumption by comparing the value of the integrand
       at four randomly chosen points with the value of the straight line interpolating the integrand at
       the two end points of the sub-interval. If test is passed, accept  $vint$ . */
135:  $slope = (fend - fbeg) * two$ 
     $fbeg2 = fbeg + fbeg$ 
    do  $i = 1, 4$ 
        if ( $inf\_integ$ ) then
             $diff = \text{abs}(f\_sub(beg + rn_i * step) - fbeg2 - rn_i * slope)$ 
        else
             $diff = \text{abs}(f(beg + rn_i * step) - fbeg2 - rn_i * slope)$ 
        end if
        if ( $tabtlm + diff \neq tabtlm$ )
            go to 155
    end do
    go to 160

```

```

/* Noise may be a dominant feature: estimate noise level by comparing the value of the integrand
   at four randomly chosen points with the value of the straight line interpolating the integrand at
   the two endpoints. If small enough, accept vint. */
145: slope = (fend - fbeg) * two
      fbeg2 = fbeg + fbeg
      i = 1
150: if (inf_integ) then
      diff = abs(f_sub(beg + rn_i * step) - fbeg2 - rn_i * slope)
      else
      diff = abs(f(beg + rn_i * step) - fbeg2 - rn_i * slope)
      end if
155: errer = max(errer, astep * diff)
      if (errer > ergoal  $\wedge$  (ergl + errer)  $\neq$  ergl)
          go to 175
      i = i + 1
      if (i  $\leq$  4)
          go to 150
      ier = 66

      /* Intergration over current subinterval successful: add vint to cadre and errer to error, then
         set up next subinterval, if any. */
160: cadre = cadre + vint
      error = error + errer
      if (right)
          go to 165
      istage = istage - 1
      if (istage  $\equiv$  0)
          go to 220
      reglar = reglsvistage
      beg = beginistage
      edn = finisistage
      curest = curest - estistage+1 + vint
      iend = ibeg - 1
      fend = tsiend
      ibeg = ibegsistage
      go to 180

165: curest = curest + vint
      stage = stage + stage
      iend = ibeg
      ibeg = ibegsistage
      edn = beg
      beg = beginistage
      fend = fbeg
      fbeg = tsibeg
      go to 5

      /* Integration over current subinterval is unsuccessful; mark subinterval for further subdivision.
         Set up next subinterval. */
170: reglar =  $\mathcal{T}$ 
175: if (istage  $\equiv$  mxstge)
      go to 205
      if (right)
          go to 185

```

```

reglsvistage+1 = reglar
beginistage = beg
ibegsistage = ibeg
stage = stage * half
180: right =  $\mathcal{T}$ 
beg = (beg + edn) * half
ibeg = (ibeg + iend) / 2
tsibeg = tsibeg * half
fbeg = tsibeg
go to 10
185: nnleft = ibeg - ibegsistage
if (iend + nnleft  $\geq$  maxts)
    go to 200
    iii = ibegsistage
    ii = iend
    do i = iii, ibeg
        ii = ii + 1
        tsii = tsi
    end do
    do i = ibeg, ii
        tsiii = tsi
        iii = iii + 1
    end do
    iend = iend + 1
    ibeg = iend - nnleft
    fend = fbeg
    fbeg = tsibeg
    finisistage = edn
    edn = beg
    beg = beginistage
    beginistage = edn
    reglsvistage = reglar
    istage = istage + 1
    reglar = reglsvistage
    estistage = vint
    curest = curest + estistage
    go to 5
/* Failure to handle given integration problem */
200: ier = 131
    go to 215
205: ier = 132
    go to 215
210: ier = 133
215: cadre = curest + vint
220: deadre = cadre
9000: continue
9005: return
end

```

Locate thresholds. This routine is designed to pin down the locations of “thresholds” in the input function. The procedure is loosely related to that of finding a root, except that the function may be discontinuous at the threshold so that a very robust (bisection) method must be used. An initial interval  $lower\_bound < x < upper\_bound$  is input. The function  $f\_of\_x$  is specified externally. The routine will return revised bounds  $lower\_bound\_rev$  and  $upper\_bound\_rev$  such that  $f\_of\_x(lower\_bound\_rev) \neq 0$ , but  $f\_of\_x(lower\_bound\_rev - \epsilon) = 0$  and  $f\_of\_x(upper\_bound\_rev) \neq 0$  with  $f\_of\_x(upper\_bound\_rev + \epsilon) = 0$ , where  $\epsilon$  is the input parameter  $error$ .

```

⟨Functions and subroutines 2⟩ +≡
subroutine find_threshold(lower_bound, upper_bound, f_of_x, error, lower_bound_rev, upper_bound_rev)
  implicit none_f77
  implicit none_f90

  real lower_bound, upper_bound, f_of_x, error // Input
  real lower_bound_rev, upper_bound_rev // Output
  real f_lower, f_upper, guess, lower_guess, /* Local */
    upper_guess, bisect
  external f_of_x // External

  f_lower = f_of_x(lower_bound)
  f_upper = f_of_x(upper_bound)

  lower_bound_rev = lower_bound // Default values
  upper_bound_rev = upper_bound

  /* If f_of_x is nonzero at one of the limits, use that limit with bisect to to find the threshold at
   the other end. Can then return. */
  if ((f_lower ≠ zero) ∨ (f_upper ≠ zero)) then
    if (f_upper ≡ zero)
      upper_bound_rev = bisect(upper_bound, lower_bound, error, f_of_x)
    if (f_lower ≡ zero)
      lower_bound_rev = bisect(lower_bound, upper_bound, error, f_of_x)
  else

    /* Zero at both limits. Since find_nonzero cannot accept a bound of zero, replace lower_bound
     with integration_acc. But, first check to see if f_of_x is nonzero there; if so, do not need to call
     find_nonzero! */
    if (lower_bound ≡ zero) then
      if (f_of_x(integration_acc) ≠ zero) then
        lower_guess = zero
        guess = integration_acc
        upper_guess = upper_bound
      else
        call find_nonzero(integration_acc, upper_bound, lower_guess, guess, upper_guess, f_of_x)
      end if
    else
      call find_nonzero(lower_bound, upper_bound, lower_guess, guess, upper_guess, f_of_x)
    end if

    /* If find_nonzero fails, it will set the upper and lower guesses equal. Otherwise, use bisect to
     find the two thresholds. */
    if (lower_guess ≠ upper_guess) then
      upper_bound_rev = bisect(upper_guess, guess, error, f_of_x)
      lower_bound_rev = bisect(lower_guess, guess, error, f_of_x)
    end if
  end if

```

```
    return  
end
```

Find a nonzero value of a function. Since nothing is assumed to be known a priori about the function  $f\_of\_x$ , this routine must be able to cover the input interval *lower\_bound* to *upper\_bound* thoroughly. For this reason, a sequence of successively finer subdivisions, much like that used to incrementally refine a trapezoidal integration, is used to break up the interval into smaller and smaller pieces. Of course, the routine is essentially done once *one* nonzero value of  $f\_of\_x$  is found. This value of  $x$  is returned as *guess*. One virtue of the approach used here is that adjacent  $x$  at which  $f\_of\_x = 0$  are already known from previous evaluations. These are returned as *lower\_guess* and *upper\_guess*.

Note that because a logarithmic subdivision is used, both of the bounds should be positive.

The parameter used to limit the number of subdivisions made is *min\_bound\_ratio*. It represents the minimum ratio of *upper\_guess* to *lower\_guess* which could be resolved by this routine. The value of *min\_bound\_ratio* should not be too small since the maximum number of function evaluations is  $2^{max\_stage} - 1$ ; i.e., a large number.

```

⟨ Functions and subroutines 2 ⟩ +≡
subroutine find_nonzero(lower_bound, upper_bound, lower_guess, guess, upper_guess, f_of_x)

implicit none f77
implicit none f90

real lower_bound, upper_bound, f_of_x // Input
real lower_guess, guess, upper_guess // Output
integer i, istage, max_stage // Local
real log_lower, log_upper, log_min_ratio, delta, log_guess

assert(upper_bound > zero)
assert(lower_bound > zero) /* Divide up the interval (in log10 space). The maximum number of
function evaluations is  $2^{max\_stage} - 1$ . */
log_upper = log10(upper_bound)
log_lower = log10(lower_bound)
log_min_ratio = log10(min_bound_ratio)
max_stage = (log((log_upper - log_lower) / log_min_ratio) / log(two)) + one

do istage = 1, max_stage

/* Calculate the step size (in log10 space) and log10 of the first x for each istage. */
delta = (log_upper - log_lower) / twoistage-1
log_guess = log_lower + half * delta

/* Each stage consists of evaluating f_of_x at only those new values of x required to achieve a
grand total of  $2^{istage} - 1$  evaluations. */
do i = 1, 2istage-1
guess = const(1., 1)log_guess
if (f_of_x(guess) ≠ zero)
    goto loop
log_guess = log_guess + delta
end do
end do

/* Cannot find a point where f_of_x ≠ 0; indicate this condition by setting lower_guess =
upper_guess. */
lower_guess = lower_bound
upper_guess = lower_guess
return

loop: continue

```

```

/* Having found a nonzero value of  $f\_of\_x$ , know that the points (in log space)  $\delta/\log(2)$  above and
   below have  $f\_of\_x = 0$ . */
upper_guess = const(1., 1) $^{\log\_guess + \frac{1}{2}\delta}$ 
lower_guess = const(1., 1) $^{\log\_guess - \frac{1}{2}\delta}$ 
return
end

```

Interval bisection routine. This function is called by *find\_threshold* to further bisect the input interval to localize the threshold to the specified accuracy. The first argument *zero\_side* is the side of the interval at which  $f\_of\_x = 0$ ; likewise, at *nonzero\_side*,  $f\_of\_x \neq 0$ . It is assumed that *find\_threshold* has arranged these in the proper order, and they are not checked here. The interval is bisected until it is *error* wide. The function returns the bound of that interval at which  $f\_of\_x \neq 0$ .

```

⟨Functions and subroutines 2⟩ +≡
function bisect(zero_side, nonzero_side, error, f_of_x)
  implicit none_f77
  implicit none_f90
  real bisect
  real zero_side, nonzero_side, error, f_of_x // Input
  integer i, ibound // Local
  real rev_zero, rev_nonzero, guess

  /* Determine number of bisections required to shrink interval to error */
  ibound = ( $\log(\text{abs}(\text{nonzero\_side} - \text{zero\_side}) / \text{error}) / \log(2)$ ) + 1

  rev_zero = zero_side
  rev_nonzero = nonzero_side
  do i = 1, ibound
    guess =  $\frac{1}{2}(\text{rev\_zero} + \text{rev\_nonzero})$ 
    if (f_of_x(guess) ≡ zero) then
      rev_zero = guess
    else
      rev_nonzero = guess
    end if
  end do

  bisect = rev_nonzero // Assign point with last nonzero f_of_x

  return
end

```

## 7 References

## References

- [1] D. Reiter, in *Atomic and Plasma-Material Interaction Processes in Controlled Thermonuclear Fusion*, R. K. Janev and H. W. Drawin, Eds. (Elsevier, New York, 1993), p. 243.
- [2] R. K. Janev, W. D. Langer, K. Evans, Jr., and D. E. Post, Jr., *Elementary Processes in Hydrogen-Helium Plasmas* (Springer-Verlag, New York, 1987).
- [3] R. K. Janev and J. J. Smith, *Atomic and Plasma-Material Interaction Data for Fusion* (Supplement to the journal Nuclear Fusion) **4** (IAEA, Vienna, 1993).
- [4] P. J. Davis and P. Rabinowitz, *Methods of Numerical Integration* (Academic Press, New York, 1984).
- [5] C. de Boor, in *Mathematical Software*, J. R. Rice, Ed. (Academic Press, New York, 1971), p. 417.

## 8 INDEX

*a:* 6.1.  
*abs:* 6.1, 6.4.  
*absi:* 6.1.  
*ACXGL1:* 5.1.  
*ACXGL2:* 5.1.  
*aerr:* 6.1.  
*AIONGL:* 5.1.  
*ait:* 6.1.  
*aitken:* 6.1.  
*aitlow:* 6.1.  
*aittol:* 6.1.  
*alad\_diskin:* 5.  
*aladdin\_eval\_fit:* 3.1, 4.4, 5.1.  
*aladdin\_read\_coefs:* 3, 5.  
*ALCHEB:* 5.1.  
*alcomp:* 5.  
*alg4o2:* 6.1.  
*ALJAN3:* 5.1.  
*ALKING:* 5.1.  
*all\_ints:* 2.2.  
*ALMEWE:* 5.1.  
*alpha:* 6.1.  
*ALPHACX:* 5.1.  
*already:* 5.  
*alrecf:* 5.  
*and\_section:* 2.3.  
*AQEXC1:* 5.1.  
*AQEXC2:* 5.1.  
*AQEXC3:* 5.1.  
*areal:* 3, 3.1, 3.2, 4, 4.4, 6.1.  
*arg:* 1.1.  
*arg\_count:* 1.1.  
*assert:* 1.1, 2.1, 2.2, 2.3, 3, 3.1, 3.2, 5, 5.1, 6, 6.3.  
*astep:* 6.1.  
*atomic\_mass\_unit:* 2.2.  
*atomic\_mass\_unit\_g:* 3.1, 4.2.  
  
*b:* 2.1, 2.2, 2.3, 3, 6.1.  
*beg:* 6.1.  
*begin:* 6.1.  
*BELI:* 5.1.  
*bisect:* 6.2, 6.4.  
*bl:* 5.  
*bool\_and\_label:* 5.  
*bool\_list:* 2.3.  
*bool\_not\_label:* 5.  
*bool\_sect:* 2.3.  
  
*cadre:* 6.1.  
*cf:* 5.  
  
*command\_arg:* 1.1.  
*const:* 2.1, 2.2, 4, 4.1, 6.1, 6.3.  
*curest:* 6.1.  
  
*dble:* 3.1, 4.4.  
*dcadre:* 6, 6.1.  
*decl\_alpcom:* 5.  
*delta:* 3, 6.3.  
*dep\_var:* 2.2, 2.3.  
*description:* 3.  
*dif:* 6.1.  
*diff:* 6.1.  
*diskin:* 2, 5.  
  
*e:* 2.1, 2.2, 2.3, 3.  
*e\_distrib\_fn:* 4, 4.1, 4.3.  
*edn:* 6.1.  
*EEXCH2:* 5.1.  
*EEXCJN:* 5.1.  
*EIONHR:* 5.1.  
*EIONH2:* 5.1.  
*EIONJN:* 5.1.  
*ELCROSS:* 5.1.  
*electron\_charge:* 2.2.  
*electron\_norm\_integrand:* 3.1, 4.1.  
*electron\_rate\_integrand:* 3.1, 4.  
*ELSPLINE:* 5.1.  
*end\_dep\_var:* 2.1.  
*end\_indep\_var:* 2.2.  
*end\_of\_file:* 5.  
*energy:* 4, 4.1, 4.3, 4.4.  
*eof:* 2, 2.3.  
*ergl:* 6.1.  
*ergoal:* 6.1.  
*erra:* 6.1.  
*errer:* 6.1.  
*erret:* 6.1.  
*errmsg:* 4.4, 5.  
*error:* 3.1, 6, 6.1., 6.2, 6.4.  
*errr:* 6.1.  
*est:* 6.1.  
*ETABSQ:* 5.1.  
*ev\_to\_ergs:* 4, 4.1, 4.2.  
*exp:* 3, 4.2, 4.3.  
*exponent:* 4.  
*exponent\_minus:* 4.2.  
*exponent\_plus:* 4.2.  
  
*f:* 6.1.  
*f\_lower:* 6.2.  
*f\_of\_x:* 6.2, 6.3, 6.4.  
*f\_sub:* 6.1.  
*f\_upper:* 6.2.  
*fbeg:* 6.1.

*fbeg2:* 6.1.  
*fend:* 6.1.  
*feutm1:* 6.1.  
*feextrp:* 6.1.  
*fi:* 6.1.  
*file:* 2, 5.  
*FILE:* 1.  
*fileid:* 3.  
*FILELEN:* 1.1, 2, 2.2, 3.  
*filename:* 1.1, 2, 5.  
*fileout:* 3.  
*fill\_tables:* 2.3, 3.  
*find\_delta:* 3, 3.2, 5.1.  
*find\_nonzero:* 6.2, 6.3.  
*find\_threshold:* 6, 6.2, 6.4.  
*finis:* 6.1.  
*fit\_coef:* 3, 3.1, 4.4, 5, 5.1.  
*fn:* 6.1.  
*FNAGEX:* 5.1.  
*FNRC:* 5.1.  
*fysize:* 6.1.  
*FNXS:* 5.1.  
*form:* 2.  
*four:* 6.1.  
*fourp5:* 6.1.  
*FORC:* 5.1.  
*FOX5:* 5.1.  
*guess:* 6.2, 6.3, 6.4.  
*half:* 4, 4.2, 6.1, 6.3, 6.4.  
*H CX1:* 5.1.  
*H CX2:* 5.1.  
*H CX3:* 5.1.  
*H CX4:* 5.1.  
*HEXCLD:* 5.1.  
*HEXC1:* 5.1.  
*HEXC2:* 5.1.  
*HEXC3:* 5.1.  
*HEXC4:* 5.1.  
*HEXC5:* 5.1.  
*hier\_label:* 5.  
*HIONN:* 5.1.  
*hovn:* 6.1.  
*hun:* 6.1.  
*h2conv:* 6.1.  
*h2next:* 6.1.  
*h2tfex:* 6.1.  
*h2tol:* 6.1.  
*i:* 2.1, 2.2, 2.3, 6.1, 6.3, 6.4.  
*ibeg:* 6.1.  
*ibegs:* 6.1.  
*ibound:* 6.4.  
*idep:* 2.2, 2.3.  
*iend:* 6.1.  
*ientry:* 5.  
*ier:* 6, 6.1.  
*ii:* 6.1.  
*iii:* 6.1.  
*implicit\_none.f77:* 1.1, 2, 2.1, 2.2, 2.3, 3, 3.1, 3.2,  
  4, 4.1, 4.2, 4.3, 4.4, 5, 5.1, 6, 6.1, 6.2, 6.3, 6.4.  
*implicit\_none.f90:* 1.1, 2, 2.1, 2.2, 2.3, 3, 3.1, 3.2,  
  4, 4.1, 4.2, 4.3, 4.4, 5, 5.1, 6, 6.1, 6.2, 6.3, 6.4.  
*ind\_var:* 3, 3.1.  
*ind\_var\_delta:* 3.  
*ind\_var\_min:* 3.  
*index:* 5.1.  
*inf\_integ:* 6.1.  
*infinite\_integral:* 3.1, 6.  
*integ\_transform\_exp:* 6.1.  
*integ\_transform\_ratio:* 6, 6.1.  
*integrand:* 6.  
*integration\_acc:* 6, 6.2.  
*integration\_infinity:* 6.  
*INTERP1D:* 5.1.  
*INTERP2D:* 5.1.  
*ion\_rate\_integrand:* 3.1, 4.2.  
*IONBEA:* 5.1.  
*istage:* 6.1, 6.3.  
*istep:* 6.1.  
*istep2:* 6.1.  
*it:* 6.1.  
*j:* 2.1, 2.2, 3, 3.1.  
*JANRD:* 3.1, 5.1.  
*JAN1:* 5.1.  
*JBORN1:* 5.1.  
*JBORN2:* 5.1.  
*JCX1:* 5.1.  
*JCX2:* 5.1.  
*JDREC1:* 5.1.  
*JEEXC1:* 5.1.  
*JEEXC2:* 5.1.  
*JEEXC3:* 5.1.  
*JEEXC4:* 5.1.  
*JEEXC5:* 5.1.  
*JEEXC6:* 5.1.  
*JEEXC7:* 5.1.  
*JEION5:* 5.1.  
*JIONHE:* 5.1.  
*JRREC1:* 5.1.  
*JRREC2:* 5.1.  
*JRREC3:* 5.1.  
*jumpl:* 6.1.  
*k:* 3.

*kernmsg:* [5.1](#).  
*l:* [3](#), [6.1](#).  
*len:* [2.1](#), [2.2](#), [2.3](#).  
*length:* [2](#), [2.1](#), [2.2](#), [2.3](#), [6.1](#).  
*leval:* [3](#), [3.1](#), [4.4](#), [5](#), [5.1](#).  
*lge:* [2.2](#).  
*line:* [2](#), [2.1](#), [2.2](#), [2.3](#).  
*LINELEN:* [1.1](#), [2](#), [3](#).  
*lle:* [2.2](#).  
*lm1:* [6.1](#).  
*log:* [3.2](#), [6.3](#), [6.4](#).  
*log\_guess:* [6.3](#).  
*log\_lower:* [6.3](#).  
*log\_min\_ratio:* [6.3](#).  
*log\_upper:* [6.3](#).  
*log10:* [6.1](#), [6.3](#).  
*loop:* [5](#), [6.3](#).  
*loop1:* [2](#).  
*lower\_bound:* [6.1](#), [6.2](#), [6.3](#).  
*lower\_bound\_rev:* [6](#), [6.2](#).  
*lower\_guess:* [6.2](#), [6.3](#).  
*lower\_result:* [6](#).  
*m:* [3](#).  
*match:* [5](#).  
*max:* [6.1](#).  
*max\_stage:* [6.3](#).  
*max\_value:* [3.2](#).  
*maxtbl:* [6.1](#).  
*maxts:* [6.1](#).  
*min\_bound\_ratio:* [6.3](#).  
*min\_value:* [3.2](#).  
*moment\_number:* [3.1](#), [4](#), [4.2](#).  
*msg\_length:* [3.1](#), [4.4](#), [5](#), [5.1](#).  
*multiplier:* [3.2](#), [4](#), [4.1](#), [4.2](#).  
*mxstge:* [6.1](#).  
*n:* [6.1](#).  
*nbl:* [5](#).  
*NC\_CHAR:* [3](#).  
*NC\_CLOBBER:* [3](#).  
*nc\_decls:* [3](#).  
*NC\_GLOBAL:* [3](#).  
*nc\_stat:* [3](#).  
*ncatputc:* [3](#).  
*ncclose:* [3](#).  
*nccreate:* [3](#).  
*ncendef:* [3](#).  
*NEEXH1:* [5.1](#).  
*NEEXH2:* [5.1](#).  
*new\_bool\_sect:* [2.3](#).  
*next\_bool:* [2.3](#).  
*next\_dep\_var:* [2.1](#), [2.2](#), [2.3](#).  
*next\_hlab:* [2.3](#).  
*next\_line:* [2.2](#).  
*next\_token:* [2.1](#), [2.2](#), [2.3](#), [3](#).  
*nnleft:* [6.1](#).  
*nonzero\_side:* [6.4](#).  
*norm\_constant:* [3.1](#).  
*num\_bool\_and:* [5](#).  
*num\_bool\_not:* [5](#).  
*num\_evaluations:* [3.1](#), [4.4](#).  
*num\_fit\_coef:* [3](#), [3.1](#), [4.4](#), [5](#), [5.1](#).  
*num\_fit\_coef\_aladdin:* [3.1](#).  
*num\_hier:* [5](#).  
*number\_of\_values:* [3.2](#).  
*n2:* [6.1](#).  
*one:* [2.1](#), [2.2](#), [3](#), [4](#), [4.2](#), [6](#), [6.1](#), [6.3](#).  
*p:* [2.1](#), [2.2](#), [2.3](#), [3](#).  
*parse\_string:* [2](#), [2.1](#), [2.2](#), [2.3](#).  
*pet:* [5.1](#).  
*PEXCH4:* [5.1](#).  
*pfit:* [5.1](#).  
*PI:* [4](#), [4.1](#), [4.2](#).  
*prever:* [6.1](#).  
*p1:* [6.1](#).  
*r:* [6.1](#).  
*ra:* [1](#), [2.3](#).  
*ra\_bool\_and\_label:* [2.3](#), [3](#).  
*ra\_bool\_not\_label:* [2.3](#), [3](#).  
*ra\_common:* [2](#), [2.3](#), [3](#), [3.1](#), [4](#), [4.1](#), [4.2](#).  
*ra\_data\_filename:* [2.3](#), [3](#).  
*ra\_data\_filename\_length:* [5](#).  
*ra\_fit\_coef\_max:* [2.2](#), [5](#), [5.1](#).  
*ra\_hier\_label:* [2.3](#), [3](#).  
*ra\_label\_max:* [5](#).  
*ra\_localcommon:* [3](#), [3.1](#), [4](#), [4.2](#), [4.3](#), [4.4](#).  
*ra\_mass:* [1](#), [2.3](#), [3.1](#), [4](#), [4.1](#), [4.2](#).  
*ra\_mass\_units:* [2.3](#), [3.1](#).  
*ra\_num:* [2](#).  
*ra\_num\_bool\_and:* [2.3](#), [3](#).  
*ra\_num\_bool\_not:* [2.3](#), [3](#).  
*ra\_num\_hier:* [2.3](#), [3](#).  
*ra\_search\_label\_length:* [5](#).  
*ratecalc:* [1.1](#).  
*react\_begin:* [2.3](#).  
*react\_done:* [2.3](#).  
*read\_dependent:* [2](#), [2.1](#).  
*read\_indep:* [2](#), [2.2](#).  
*read\_input:* [1.1](#), [2](#).  
*read\_integer:* [2.2](#), [2.3](#).  
*read\_reaction\_list:* [2](#), [2.3](#).  
*read\_real:* [2.2](#), [2.3](#).  
*read\_string:* [2](#), [2.1](#), [2.2](#), [2.3](#).

*read\_text:* 2.1, 2.2, 2.3.  
*real\_unused:* 3.  
*REFL1:* 5.1.  
*REFL2:* 5.1.  
*REFL3:* 5.1.  
*regular:* 6.1.  
*reglsv:* 6.1.  
*rel\_velocity:* 4.2.  
*rerr:* 6.1.  
*rev\_nonzero:* 6.4.  
*rev\_zero:* 6.4.  
*rf\_common:* 3.  
*right:* 6.1.  
*rn:* 6.1.  
*sigma:* 4, 4.2, 4.4.  
*sing:* 6.1.  
*singnx:* 6.1.  
*slope:* 6.1.  
*SP:* 5.  
*spacing:* 3.2.  
*SPTEK:* 5.1.  
*SPTH:* 5.1.  
*sqrt:* 3.1, 4.1, 4.2.  
*st\_decls:* 2, 2.1, 2.2, 2.3, 3.  
*stage:* 6.1.  
*status:* 2, 5.  
*step:* 6.1.  
*stepmn:* 6.1.  
*stepnm:* 6.1.  
*string\_length:* 3.  
*sum:* 6.1.  
*sumabs:* 6.1.  
*sy\_decls:* 1.1.  
*t:* 6.1.  
*table\_calc:* 3.1.  
*table\_value:* 3, 3.1.  
*tabs:* 6.1.  
*tablm:* 6.1.  
*tc:* 2.2.  
*temperature:* 3.1, 4.3.  
*ten:* 6.1.  
*ts:* 6.1.  
*two:* 3.1, 4, 4.1, 6.1, 6.3, 6.4.  
*unit:* 2, 2.1, 2.2, 2.3, 5.  
*up:* 3.1, 4.2.  
*upper\_bound:* 6.1, 6.2, 6.3.  
*upper\_bound\_rev:* 6, 6.2.  
*upper\_guess:* 6.2, 6.3.  
*upper\_result:* 6.  
*var\_delta:* 3.2.  
*var\_free:* 3.  
*var\_min:* 3.2.  
*vint:* 6.1.  
*x:* 6.1.  
*xs:* 1, 3.1.  
*xs\_common:* 2, 2.1, 2.2, 2.3, 3, 3.1.  
*xs\_data:* 3.  
*xs\_data\_tab:* 3.  
*xs\_data\_table:* 3.  
*xs\_decls:* 2, 2.1, 2.2, 2.3, 3, 3.1.  
*xs\_dep\_var\_max:* 2.1, 2.2, 2.3, 3.  
*xs\_eval\_name:* 2.1, 2.2, 3, 3.1.  
*xs\_eval\_name\_length:* 5, 5.1.  
*xs\_max:* 2.1, 2.2, 3.  
*xs\_min:* 2.1, 2.2, 3.  
*xs\_mult:* 2.1, 2.2.  
*xs\_multiplier:* 2.  
*xs\_name:* 2.3, 3.  
*xs\_ndecl:* 3.  
*xs\_ncdef:* 3.  
*xs\_ncwrite:* 3.  
*xs\_num\_dep\_var:* 2, 2.1, 2.2, 2.3, 3.  
*xs\_ragged\_alloc:* 3.  
*xs\_rank:* 2, 2.1, 2.2, 3.1.  
*xs\_spacing:* 2.1, 2.2, 3.  
*xs\_tab\_index:* 2, 2.1, 2.2, 3.  
*xs\_table:* 2, 3.  
*xs\_table\_rank\_max:* 2.1, 2.2, 3, 3.1.  
*xs\_tag\_string\_length:* 3.2.  
*xs\_units:* 2, 2.1, 2.2, 3.1.  
*xs\_var:* 2.1, 2.2, 3.1.  
*xsection\_version:* 2.3, 3.  
*zbracket:* 4.2.  
*zdpsigma:* 3.1.  
*ze:* 4.2, 4.4.  
*ze\_dist:* 4, 4.1.  
*zenergy:* 3.1.  
*zero:* 2.1, 2.2, 3, 3.1, 3.2, 6, 6.1, 6.2, 6.3, 6.4.  
*zero\_side:* 6.4.  
*zfit:* 4.4.  
*zint:* 4, 4.1.  
*zsigma:* 3, 3.1, 4, 4.2.  
*zte\_aladdin:* 3.1.  
*ztemperature:* 3.1.

⟨ Functions and subroutines 2, 2.1, 2.2, 2.3, 3, 3.1, 3.2, 4, 4.1, 4.2, 4.3, 4.4, 5, 5.1, 6, 6.1, 6.2, 6.3, 6.4 ⟩ Used in section 1.1.

⟨ Memory allocation interface 0 ⟩ Used in section 3.

**COMMAND LINE:** "fweave -f -i! -W[ -ybs15000 -ykw800 -ytw40000 -j -n /Users/dstotler/degas2/src/ratecalc.web".

**WEB FILE:** "/Users/dstotler/degas2/src/ratecalc.web".

**CHANGE FILE:** (none).

**GLOBAL LANGUAGE:** FORTRAN.